A REST Service Framework for RaaS Clouds
Li Li, Wu Chou, and Min Luo
Huawei
{li.nj.li, wu.chou, min.ch.luo}@huawei.com

Abstract
RaaS (Resource-as-a-Service) clouds represent a new market and technology driven paradigm shift from renting prebuilt virtual machines towards renting resources to compose elastic machines. By offering cloud users fine-grained resources, including CPU, memory, disk and network based on their demands, RaaS clouds have the potential to address several inherent problems in IaaS clouds. RaaS architecture requires a new service layer to abstract and compose heterogeneous and dynamic fine-grained cloud resources at high frequency. This paper proposes a REST service framework consisting of Resource-Oriented Network (RON) and monadic service composition. RON abstracts heterogeneous cloud resources with uniform REST resources and represents their dynamic relations with hypertext. Monadic service composition provides a concise functional programming language with monad to concurrently control large scale RON through dynamically generated workflows. A prototype system has been implemented based on Docker containers and Linux control groups. Our experimental results show that the approach is feasible and the performance is consistent with the client and server workloads.

Keywords: RaaS, REST, Resource-oriented Network, elastic machine, monad, service composition, Docker, control group

1. Introduction

Conventional IaaS clouds are based on Virtual Machine (VM) technologies that emulate physical machines. An IaaS cloud provider offers its users a dozen or so VM types. Once the type of a VM is chosen, it will have fixed amount of CPU, memory, and I/O capacities that cannot be changed easily by the user. A user typically rents a VM for hours and days, and the cloud providers price VM and charge the users by minutes or hours. IaaS clouds based on fixed VM and long rental intervals are neither optimal for cloud providers nor flexible enough for cloud users, because in most cases the workloads and resource demands of a VM are unpredictable.

Resource-as-a-Service (RaaS) (Ben-Yehuda 2014) is a cloud computing paradigm to capture the current trends from renting virtual machines with prebuilt resources towards renting resources to compose elastic machines with variable capacities. In RaaS Cloud, fine-grained resources, like memory, CPU, and storage, will become the basic units that a cloud user can rent from a cloud provider at short time intervals. Users can dynamically mix and match different amount of fine-grained resources and change them on demand for an elastic machine. Several public cloud providers, including Amazon, Google, CloudSigma, Gridspot, and ProfitBricks, are moving towards this direction, by letting the users customize the capacities of their VMs, while pricing and charging the resources by minutes. As this trend continues, the future RaaS clouds may allow users to sell and buy fine-grained and fine-timed resources, such as 1GB memory for 10 seconds at $0.05, through automated economic agents. RaaS clouds can significantly increase resource utilization for the providers, and at the same time, decrease the rental cost for cloud users, without compromising the performance of cloud applications.

Besides the economic and market forces mentioned in (Ben-Yehuda 2014), several technological forces are driving cloud computing towards RaaS. The first one is the container-based virtualization, accelerated by the advance of Docker (Docker 2015) technologies that make the containers easy to build, deploy, control, and manage. The second one is the advances in network architectures, including SDN and NFV, which provide flexible control over fine-grained network resources. The third one is the disaggregated datacenters where fine-grained resources can be shared and composed from resource pools.

Docker container is much more efficient than VM, because it eliminates the hypervisor and the guest OS layers in VM, by having the container and the host share the Linux kernel while keeping them separated using Linux namespaces. A recent study (Russell 2015) shows that Docker containers consume only 6% of CPU and 16% of memory comparing to a KVM based VM, and moreover, its boot time is only 60% and its reboot time is only 2% of the corresponding VMs. Docker image size is typically in the MB range, whereas VM images are typically in the GB range. In addition, workload tests show that processes running inside Docker containers have almost the same performance as running on host machines, making containerization more efficient than virtualization for cloud computing.

The significant improvement of container over VM is not limited in performance but also in the deployment density. The small footprint of container leads to a deployment density that can be 6-12 times of the deployment density of VMs. High deployment density in cloud computing means that fewer physical machines are needed to support the same amount of clients, thereby
reducing the capital (e.g. purchase) and operation (e.g. energy) costs for cloud providers. For these reasons, many companies and open source communities, including Google, IBM, Microsoft, VMware, and OpenStack, are investing in containerization technologies for cloud computing, and the advance of Docker makes cloud computing at the Internet scale a potential reality.

Recent advances in network architectures can be regarded as following a trend towards RaaS. In particular, the combination of SDN and NFV are opening datacenter networks to the cloud users as a programmable platform.

Software-Defined Networking (SDN) (ONF 2015) decouples the data plane and control plane that were tightly coupled and distributed in traditional network architecture. This separation enables a logically centralized SDN controller to control an entire network of devices through some southbound APIs and to provide a northbound REST API for users to create, configure, and control a virtual network over the HTTP protocol.

Network Functions Virtualization (NFV) (NFV 2015) moves various network functions, such as NAT and FW, from proprietary hardware boxes to commodity servers, to reduce cost and increase flexibility. Virtual network functions from different vendors can run on a common datacenter network platform. Users can rent network functions and compose them to process network flows according to their business needs.

Disaggregated datacenters can radically change cloud computing, by allowing a virtual machine to access fine-grained resources drawn from some large resource pools instead of from the servers. In addition, while storage has been disaggregated from CPU and memory by technologies such as NAS (Network Attached Storage) and SAN (Storage Area Network), recent work (Lim 2009, Han 2013, Weiss 2014, Li, C.S. 2015, Yoshikawa 2015) also explored the disaggregated CPU and memory, such that a CPU core in a pool can be composed with memory frames in another pool within the limited delay.

Disaggregated datacenter aims to eliminate resource fragmentations inherent in server-based datacenters and RaaS is a natural fit for that, because they share the same goal that machines are not prebuilt but dynamically composed on demand. We think the central idea of RaaS can be embodied by Elastic Machine (EM) described in this paper, which can take the form of VM, Docker container or other types of machines, and it has the following properties:

- An EM can be composed from cloud resources (distributed or collocated) on demand.
- The cloud resources of an EM can change over time at high frequency.
- An EM is autonomous and can trade cloud resources with the provider or another EM.
- An EM can be composed from other EMs.

To realize RaaS based on the concept of EM requires a new service layer to abstract and compose fine-grained and fine-timed cloud resources, and the service layer need to address the following challenges:

1. Heterogeneity: how to deal with variations in space, i.e. abstract the wide range of fine-grained resources (e.g. CPU, GPU, FPGA and ASIC) that differ in many dimensions, including types, functions, sizes, access methods, and communication protocols, into some uniform services for the clients.
2. Dynamism: how to deal with variations in time, i.e. hide from clients the changes to fine-grained resources due to hardware, firmware, and software updates.
3. Composability: how to maintain and represent various and dynamic organizations of cloud resources in an EM by a uniform, extensible, and easy to understand way for the clients (for example, an EM connects several hybrid ARM64 cores with the same instruction set but different microarchitectures to remote memory).
4. Overhead: how to minimize the potential overhead introduced by the service layer to the communications between the fine-grained resources, when they have strict latency requirements (for example, the 20ms latency requirement for CPU-Memory communication).
5. Scalability: how to use small programs to concurrently and efficiently control 10^5 to 10^6 fine-grained resources in a datacenter, which has 10^3 or more EMs, and each EM can have 10-100 fine-grained resources.

Heterogeneity of cloud resources can come from two sources: intrinsic and configuration variations. Fine-grained cloud resources differ in types, functions, interfaces, performances, and costs. For example, computing resources include CPU, GPU, FPGA and ASIC, etc., and memory resources can be volatile or nonvolatile, physical or virtual, and organized in UMA or NUMA architectures. The physical servers in a datacenter may be configured to run different operating systems or the same operating system with different services.

Dynamism of cloud resources can come from two sources: evolutions by design and changes by configurations. As the Linux operating system undergoes updates and redesigns, the APIs to access the fine-grained resources can also change over time. The recent redesign of the cgroup hierarchy (Heo 2014), should it be widely adopted, will alter the current Linux cgroup structure. As fine-grained resources are added or removed by operations from the clients, the topology of the RON can change in some significant ways as well.

To address these challenges, this paper proposes a REST service framework that consists of Resource-Oriented Network (RON) and monadic service composition. In particular, RON addresses the first four challenges, while monadic service composition addresses the last challenge. RON abstracts heterogeneous and dynamic cloud resources
and their relationships as connected and uniform REST services. Monadic composition provides a concise functional programming language with monad to concurrently and efficiently discover and invoke the REST services in RONs through dynamically generated workflows.

This framework has been combined with Docker container technology, where Docker and its REST API provide a basic construct of computing unit, and our REST framework provides an efficient and extensible architecture to dynamically manage the resources allocated to Docker containers.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the proposed RON framework and its architectural properties, and Section 4 describes some concrete RON REST services and the solutions to address the above challenges. Section 5 describes monadic service composition language and its runtime architecture. Section 6 describes the implementation and experimental results, and we conclude the findings of this paper with Section 7.

2. RELATED WORK

2.1 FINE-GRAINED RESOURCE CONTROL

Figure 1 is a high-level architecture of Docker, where a Docker daemon provides REST APIs to launch and manage Docker containers from Docker images that package the additional libraries and files needed by the applications. Docker containers can run on any Linux flavored platforms as long as they have the required Linux kernel. A Docker container has its own Linux file system and process namespace isolated from the host Linux OS and other containers. To users, a Docker container is like a VM. For example, we can run an Ubuntu container on a CentOS host. Docker uses Union File System to organize its images so that a Docker image can be updated by just downloading the added layer to bridge the difference.

Table: Docker Architecture

<table>
<thead>
<tr>
<th>Applications (Web server, database, etc.)</th>
<th>Docker Containers</th>
<th>Docker Daemons (REST API)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux OS (Redhat, CentOS, Ubuntu, Fedora)</td>
<td>Hardware (CPU, memory, storage, network)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Docker architecture

Docker uses Linux control groups (cgroups) (Cgroup 2015) to allocate hardware resources among the containers, such that the capacity of each container can be adjusted dynamically using the cgroup API. For example, we can change the memory limit or the number of CPU cores of a running container at any time. Since cgroup is a Linux kernel feature that works for any processes, it is possible to resize the capacity dynamically to meet the application needs. However, as far as we know, cgroup has only a command-line interface and there is no REST API to cgroups, making it difficult to manage the fine-grained resources as services.

Most hypervisors, such as KVM, vSphere, Xen, and Hyper-V, provide a control interface similar to the one in (VMWare 2015) for dynamically adjusting the capacity of a VM, including its CPU time, CPU cores, and memory size. But not all guest OS can take advantage of the increased capacity in the same way as containers, and some tests (Rao 2009, 2011) show that a VM takes minutes to become stable after its capacity is changed. Docker avoids this issue because it is container based. A container does not involve any hypervisor or guest OS in its operation, and its running resource state can be changed almost instantly.

There are several computer cluster management systems based on Docker technologies. Kubernetes (Kubernetes 2015) is an open-source project from Google that provides REST API to deploy and manage tightly integrated containers (called Pod) in clusters. But it can only control a Pod as the smallest unit. Apache Mesos (Mesos 2015) is an open source cluster scheduler that collects fine-grained computing resources, e.g. CPU and memory, in a cluster and offers them to the registered frameworks which in turn will schedule their tasks based on the offers. However, Mesos does not provide a REST API to control those fine-grained resources while the tasks are running.

Docker has been integrated into OpenStack (OpenStack 2015), an open-source IaaS cloud platform, where Docker containers can run on top of VMs. However, OpenStack currently does not provide REST API to dynamically reconfigure container’s capacity. CoreOS (CoreOS 2015) is a Linux OS customized for Docker containers and it provides API and job scheduler to deploy and manage containers on the CoreOS hosts. However, the smallest control unit in CoreOS is a container, and it does not control fine-grained resources within a container.

2.2 SERVICE COMPOSITION

WS-BPEL (Jordan 2007) is a standard XML based service composition language. Its primitives include basic activities for service interactions, such as <invoke> that sends a message to a service, <reply> that receives a message from a service, and structural activities that control these interactions. Structural activities include <sequence> that executes the child activities sequentially, <flow> that executes the child activities concurrently, and <pick> that waits for exactly one from the child activities. However, it falls short to support runtime generation of workflows:

- The structural activities in WS-BPEL are static because its child activities must be fully specified at the design time. This restriction makes it difficult to
handle situations where the resource identifications and operations are unknown at design time.

- WS-BPEL depends on WSDL, but most REST services do not use WSDL, because WSDL groups operations by centralized interfaces, whereas REST services group operations by distributed hypertext.
- WS-BPEL uses complicated correlation set to identify process sessions, but this is incompatible with REST services that use URI to identify sessions.
- From the performance and portability perspectives, using a WS-BPEL engine this way is an over-kill as most of its features are not directly applicable in the paradigm of REST services.

On the other hand, WS-BPEL can define abstract process, which is a template with opaque or omitted activities. These opaque and omitted activities in WS-BPEL can be instantiated and extended by different executable processes. Although this is a very useful feature to reuse an abstract process for different purposes, it lacks a way to compose abstract and executable processes. In fact, abstract processes cannot be embedded in executable processes by definition, making it difficult to use in practice.

BPEL for REST (Pautasso 2008) is an effort to adapt WS-BPEL to REST service composition by adding a set of REST service related activities to WS-BPEL, including <Get>, <Put>, <Post>, <Delete>, and <onGet> that are modeled after HTTP 1.1 operations. Bite (Rosenberg 2008) is another REST service composition language modeled after WS-BPEL, but it is much more lightweight than WS-BPEL. Bite encodes a service composition as a flat graph that consists of activities and links between them. The activities include HTTP communications, utilities, and control primitives, while the links are URI templates that can change dynamically. Communication activities can be executed concurrently if there is no dependency between them. Bite treats a composition process itself as a REST resource that can be created, inspected, and managed with HTTP.

JOpera (Pautasso 2009) provides a GUI for creating a service composition with control and data flow graphs, where nodes in a control flow represent tasks that invoke REST services or process internal data. A task can contain variable parameters, such as URI, method, and body, to be substituted at runtime. The tasks in JOpera are dynamically bound to runtime programs by various adaptors, such that a task can include Unix shell, XSLT, and Java. JOpera also treats a workflow as a regular REST resource like Bite.

Petri-Net models have also been applied to create and analyze service compositions in several research work (Hamadi 2003, Decker 2008, Xiong 2010, Alarcon 2011). In a service composition, Petri-Net transitions represent activities that operate on tokens (messages and data) stored in the places, while the entire workflow is defined by a Petri-Net structure. Petri-Net is a powerful mathematical model that can represent concurrency, synchronization, and choices, but it lacks primitives to interact with REST services, or rules that govern the compositions of Petri-Nets.

Although some of these languages provide templates to facilitate dynamic service compositions, they lack a concise and formal method to compose the templates, or abstract a workflow for different applications.

FP System (Backus 1977) is a functional programming language proposed by John Backus in his Turing Award Lecture. The FP language makes extensive use of lists and algebraic rules to compose functions, such that a short program can represent a complex process. Monad (Wadler 1992) is a functional programming construct to encapsulate computations with side-effects, such as I/O and state, and allow them to be manipulated with functions. A monad \( M \) is defined as an operator \( M \) on type \( x \), together with three functions:

1. \( \text{map}: (x \to y) \to (M x \to M y) \): transform monad \( M x \) to monad \( M y \) with a function \( x \to y \).
2. \( \text{unit}: x \to M x \): construct monad \( M x \) from type \( x \).
3. \( \text{join}: M M x \to M x \): flatten a nested monad.

The above functions must satisfy the following rules, where \( id \) denotes the identity function that maps anything to itself, \( f, g \) are functions, and operator \( \circ \) composes functions:

- \( \text{map}(id) = id \);
- \( \text{map}(g \circ f) = \text{map}(g) \circ \text{map}(f) \);
- \( \text{map}(f) \circ \text{unit} = \text{unit} \circ f \);
- \( \text{map}(f) \circ \text{join} = \text{join} \circ \text{map}(\text{map}(f)) \);
- \( \text{join} \circ \text{unit} = id \);
- \( \text{join} \circ \text{map}(\text{unit}) = id \);
- \( \text{join} \circ \text{join} = \text{join} \circ \text{map}(\text{join}) \).

The diagram in Figure 2 illustrates how these functions are connected by these rules. For brevity, the id functions that loop back on each box are not shown. By these rules, we can transform the typed data inside monads using functions.

![Figure 2. Illustration of Monads Laws](image-url)

Taking rule c) as an example: it says that if \( M x \) is a monad constructed from \( x \) by \( \text{unit} \), and \( f \) is a function that maps \( x \) to \( y \), then we can apply \( f \) to \( M x \) to produce a monad \( M y \), by first applying \( f \) to \( x \) to produce \( y \), then constructing the monad \( M y \) from \( y \).

The map function is often used to apply a function to a list of items, which is a basic monad. Given a list monad:

\[ \text{List} \text{ int} = \{1, 2, 3\}, f(x)=2x \text{ and } g(x)=x-1 \], we have:
The `join` function can flatten the nested list monads, for example:

\[ \text{join}([[[1, 2, 3], [4, 5]], [[6, 7, 8], [9, 10]]]) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}. \]

A monad can also be defined by a `bind` function: `bind:\text{bind}(M x) = \text{map}(f)(M x)` to `M y`, such as function `h` in Figure 2, to produce monad `M y`. The `bind` and `map` functions are related by the equations:

\[ \text{h} = \text{unit}\circ f \text{ and } \text{bind}(M x, h) = \text{map}(f)(M x) \]

Looking at the diagram in Figure 2 from left to right, we can regard monad `M` as a container that carries a work item (e.g. `x`) through a pipeline. The `unit` function loads the work items into the containers, and the `map` and `join` functions act as a controller that calls a worker function (e.g. `f`) to process the work item to produce a new container.

By separating the container, controller, and workers, the monadic design offers the following advantages, in addition to the benefits of the functional programming: 1) modularity: pipelines can be constructed from containers, controllers, and workers; 2) flexibility: a pipeline can be changed easily by replacing its components; and 3) isolation: impure functions whose output depending on hidden states can be encapsulated in monads and safely incorporated into a pure functional programming language.

3. Resource-Oriented Network Model

Instead of replacing IaaS by RaaS, we take an incremental approach and insert RaaS as a new service layer between IaaS and the operating systems, as illustrated in Figure 3, where each layer is depicted as a rectangle. In this new cloud stack, RaaS exposes fine-grained resources as RON (Resource-Oriented Network) and EM (Elastic Machine). A RON defines the fine-grained resources of an EM and provides REST API to control and monitor the states and connections of these resources. An EM is also treated as a REST resource and can be part of a RON. At a higher level, a RON can be regarded as a dynamic graph of REST resources connected by hyperlinks, such that a REST client can reach all the REST resources from a distinct REST resource called entry point. With this abstraction, a RaaS cloud can be regarded as a dynamic graph of RONs connected by hyperlinks to the entry points.

Inspired by SDN that decouples a network into control plane and the data plane, RON decouples RaaS into three planes as shown in Figure 4:

- **Data**: It consists of heterogeneous cloud hardware and software components, such as CPU, memory, storage, network, switches, routers, etc., that communicate in a variety of protocols.
- **Control**: It consists of connected REST resources implemented in any programming languages that expose uniform interfaces to clients and interact with the cloud resources through their APIs.
- **Representation**: It consists of hypertexts exchanged between clients and RON through protocols (e.g. HTTP) that represent the states and connections of the REST resources in the control plane.

The separation of control and data planes can be logical if the REST resources are part of the components. The control and representation planes constitute the REST API of a RON. A RON is designed to have the following properties:

- **Access Transparency**: The REST resources at the control plane provide uniform interfaces (e.g. HTTP operations GET, PUT, POST and DELETE with hypertext encoded in XML or JSON) to access heterogeneous cloud resources at the data plane.
- **Location Transparency**: The REST resources at the control plane are identified by URI, which is independent of the physical locations or network addresses (MAC or IP) of the cloud resources at the data plane.
- **Connection Transparency**: The topology of a RON can change at runtime in an unconstrained way and a client can use hypertext-driven navigation to discover the connections from an entry point to the RON.
- **Control Transparency**: The communications between the cloud resources at the data plane can be
The Composability issue is addressed by Connection Transparency. When a CPU core is connected to some remote memory board at the data plane, the physical connection results in a logical composition between the CPU resource and the memory resource at the control plane. In the representation plane, the logical composition is represented by a hyperlink to the memory resource included in the hypertext retrieved from the CPU resource. Such compositions between the REST resources can be formed automatically using the Categorial Link mechanism described in (Li 2014). The hyperlink can be annotated with attributes such as bandwidth, delay, and rental duration that are useful for the clients to decide what workload can be assigned to the CPU and memory.

The Heterogeneity issue is addressed based on Access Transparency and Location Transparency, which will be discussed in sections 4.1 and 4.2. The Dynamism issue is addressed by a combination of several techniques based on the RON model, which will be discussed in section 4.3. The Scalability issue is addressed by partitions of cloud resources into small RONs, which will be discussed in sections 4.1 and 4.2, and the monad service composition, which will be discussed in section 5.

To induce the above properties of RON, we use REST Chart (Li 2011), a Petri-Net based modeling framework, to describe the REST API for a RON. The REST Chart models a REST API as a Colored Petri Net where the places define the possible resource representations of the REST API and a transition binds a hyperlink template between two places to a network protocol. The interactions between a REST Client and a REST API are modeled by tokens moving in the Petri-Net defined by the REST Chart. A key distinction between REST Chart and the previous approaches is that REST Chart follows a bottom-up design to REST API. REST Chart only specifies the resource representations and interactions at the design time, and the clients can discover the resource identifications and connections at the runtime through hypertext-driven navigation. Consequently, a REST API can change its identifications and connections without breaking the clients.

4. RON ORGANIZATIONS

The described RON framework is used to design and implement REST APIs to dynamically control the fine-grained resources of containers based on Linux Control Groups (cgroups). Due to the large number of fine-grained resources in a datacenter, it is impossible to organize them into a single RON. Moreover, a single RON is also difficult to isolate the users and providers who have different logical views over the shared cloud resources.

For these reasons, we develop 4 types of RONs described below:

- **Task RON (T-RON):** A Task EM consists of a user's container and it exposes a REST API to dynamically control the capacity of the Task EM.
- **Job RON (J-RON):** A Job EM consists of a group of related Task EMs which may be collocated or distributed, and it exposes a REST API to control the capacities of the Job EM and the Task EMs.
- **Server RON (S-RON):** A Server EM consists of aggregated or disaggregated hardware resources, and it exposes a REST API to control the capacity of the Server EM (notice that an aggregated server is not truly elastic, but a disaggregated server is).

![Figure 5. Workflow of RC-Java tool](image-url)
• Cluster RON (C-RON): A Cluster EM consists of a
group of Server EMs whose resources are shared as a pool,
and it exposes a REST API to control the capacity of the
Cluster EM and the Server EMs.

S-RON is modeled closely after cgroup hierarchy to
provide full access to its functionalities, whereas the other
RONs adopt different models from the cgroup hierarchy to
hide the underlying cgroup variations and changes from the
clients.

This RON organization has several advantages. First, it
allows the EMs to autonomously manage and trade their
cloud resources without involving the provider. Second, it
allows a client to manage the fine-grained resources at
different scales using a single RON. For example, a client
can manage the resource pool of a container using a T-RON,
and manage the resource pools of all the containers using
the service composition at a J-RON. The same is true for the
S-RON and C-RON. Third, it can provide crash fault-
tolerance against C-RON crashes. The cost of
resource pools of the J-RON by running a service
tolerance by eliminating single point of failures. Since a J-
RON is connected to all its T-RON through hyperlinks, and
a client can discover these hyperlinks at runtime following
hypertext-driven navigation, from which a client has two
control paths to each T-RON: direct and indirect through J-
RON. If the J-RON fails, the client can still control the
resource pools of the J-RON by running a service
composition on the direct links. Similarly, it can provide
fault-tolerance against C-RON crashes. The cost of
hyperlink discovery can be reduced by using page-based
link-based differential caches (Zhou 2014) at proxies or
clients that are fully compatible with HTTP.

Unlike process-centric S-RON, T-RON provides a task-
centric view on resource management, where the fine-
grained resources (CPU, memory, IO, etc.) of a task is
represented as a network REST resources. The REST API
of T-RON is summarized in Table II, where \{control\}
represents a type of fine-grained resource allocated to the
task.

A T-RON automatically discovers all the tasks from the
cgroup hierarchy on the server and builds a control model
that maps the URIs in Table II to the cgroup nodes as
illustrated in Figure 6. Upon receiving a request, its REST
API calls the appropriate Linux cgroup API based on the
control model. For example, to access the CPU shares of
task 2 of job 1, it can use URI:

\texttt{/jobs/1/tasks/2/cpu/cpu.shares;}

which is mapped to a cgroup node by the control model. As the control model may differ
on different Linux systems, a T-RON hides the cgroup
hierarchies from the clients.

4.2 JOB RON AND CLUSTER RON

An architecture that layers J-RON on T-RON and C-
RON on S-RON are illustrated in Figure 7, where 3 servers
(A through D) run two jobs (1 and 2) and 1 cluster that
consists of servers B and C. J-RON 1 for job 1 consists of
task 11 on server B and task 12 on server C, while J-RON 2
for job 2 consists of task 21 on server B and task 22 on
server C. For security reasons, J-RON 1 can only access
containers 11 and 12, while J-RON 2 can only access
containers 21 and 22. However, C-RON 1 is permitted to
access the S-RONs on server B and server C with the
administrator privilege on those servers.

<table>
<thead>
<tr>
<th>Table I. Namespaces of Server RON</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI template</td>
</tr>
<tr>
<td>/servers/{sid}/em/ron</td>
</tr>
<tr>
<td>{/cgroup}</td>
</tr>
<tr>
<td>PUT</td>
</tr>
<tr>
<td>{/cgroup}/{parameter}</td>
</tr>
<tr>
<td>PUT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II. Namespaces of Task RON</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI template</td>
</tr>
<tr>
<td>/jobs/{jid}/tasks/{tid}/em/ron</td>
</tr>
<tr>
<td>{/control}</td>
</tr>
<tr>
<td>PUT</td>
</tr>
<tr>
<td>{/control}/{parameter}</td>
</tr>
<tr>
<td>PUT</td>
</tr>
</tbody>
</table>

Figure 6. Mapping S-RON and T-RON to cgroups.

4.1 SERVER RON AND TASK RON

Each physical server can have one S-RON and several
T-RONs. Figure 6 illustrates how S-RON and T-RON are
mapped to the cgroups in a server A, where the cgroup
hierarchy is depicted as nested boxes.

S-RON provides control to the fine-grained resources of
Linux processes on a server with the REST API described in
Table I. For example, the REST API of S-RON can be used
to control the CPU shares, memory limit, and I/O rate of the
processes in container C1 with the following URIs:

\texttt{http://localhost:8080/}

\texttt{/servers/{sid}/em/ron} |

\texttt{/cgroup} |

\texttt{/parameter} |

\texttt{GET} |

\texttt{PUT} |

\texttt{GET} |

\texttt{PUT} |

\texttt{GET} |
The REST APIs of J-RON and C-RON are summarized in Table III and Table IV respectively, where the two REST APIs are very similar to each other, and they are also similar to the T-RON REST API in Table II, except the entry URIs. This similarity gives the REST APIs a familiar look-and-feel for users on one hand, and makes the REST APIs easy to create and maintain for developers on the other hand. These benefits are the results of adopting a common RON model for REST API design.

Each cgroup parameter \( p \) identified by URI template \( {.../\{\text{parameter}\}} \) at J-RON (C-RON) aggregates the values of the corresponding parameters in T-RON (S-RON) into a range \( p = (\max, \min) \), such that the clients knows the range of possible resources in this J-RON (C-RON). A J-RON (C-RON) can also run workflows to control multiple fine-grained resources, e.g., simultaneously increase the CPU usage shares of containers 11 and 12.

### 4.3 RON DYNAMICS

We employ three techniques to minimize the impact of RON changes to the REST clients: 1) dynamic mapping; 2) hypertext-driven navigation; and 3) event-driven synchronization.

Dynamic mapping can hide the underlying cgroup changes from RON, by using a control model that maps the REST resources in a RON to a given cgroup hierarchy. When the cgroup hierarchy changes, we can replace the mappings without changing the RON. This idea is illustrated in Figure 8, where a hierarchical RON for a container at the middle chooses one of the two mappings: one for unified cgroup hierarchy and one for multi cgroup hierarchy. A unified cgroup hierarchy is shown on the upper left and the mappings from the RON to the cgroup are shown by the dashed arrows from the RON nodes to the left cgroup nodes. A multi cgroup hierarchy is shown on the upper right and the mappings from the RON to the cgroup are shown by the dashed arrows from the RON nodes to the right cgroup nodes.

Despite dynamic mappings, changes to RON connections are sometimes inevitable and a client needs to use hypertext-driven navigation to invoke the same REST services despite these changes. An example of this approach is illustrated in Figure 9, which shows two different Server RONs at multi cgroup hierarchy (a) and unified cgroup hierarchy (b).

The REST interfaces and hyperlinks of the nodes in these two RONs remain the same, while the connections between some nodes are changed. For example, the path to node \( A \) in the first RON is \( \text{entry} \rightarrow \text{cgroup} \rightarrow \text{cpu} \rightarrow A \), while it becomes \( \text{entry} \rightarrow \text{root} \rightarrow A \) in the second RON. By using hypertext-driven navigation, a client that can find \( A \) in the first RON can start from the entry URI to the second RON. A more detailed example can be found in (Li 2016).

Hypertext-driven navigation is flexible but has certain cost associated with it. Since each discovery requires two messages, traversing a discovery path of \( N \) resources requires \( 2N \) messages. To address this issue, event-driven synchronization is used to proactively push changes at the Task RONs to a Job RON and from the Server RONs to a
Cluster RON. Since each event notification requires one message, detecting changes in a path of N resources requires only \( N-1 \) (event messages) + 2 (discovery messages) = \( N+1 \) messages. In the reverse direction, the data plane can notify changes to the control and representation planes based on the event architectures and techniques discussed in (Li 2010a, 2010b).

4.4 RON SECURITY ARCHITECTURE

A RON permits two kinds of actions as defined below:

- invoke: invoke the REST API as a client; and
- manage: control the lifecycle of the RON, including deploy, start, stop, pause, and resume.

Different subjects in the cloud are granted with different action permissions to a RON, and these permissions are summarized in Table V.

<table>
<thead>
<tr>
<th>Access</th>
<th>J-RON</th>
<th>T-RON</th>
<th>S-RON</th>
<th>C-RON</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>invoke</td>
<td>manage</td>
<td>invoke</td>
<td>invoke</td>
</tr>
<tr>
<td>Provider</td>
<td>manage</td>
<td>manage</td>
<td>invoke</td>
<td>manage</td>
</tr>
</tbody>
</table>

Table V. Access Policy for RON

This policy permits the cloud users to use the J-RON and T-RON REST APIs, but not to manage them. The management action is left to the cloud providers. The S-RON and C-RON are infrastructure resources which are not accessible to anyone but the cloud provider. Under this general policy, we ensure that each user can only access the J-RON associated with his jobs and T-RON associated with his tasks. If user \( u_1 \) owns job \( j_1 \) and its tasks, and user \( u_2 \) owns job \( j_2 \) and its tasks, then each user can only access his own namespaces shown in Table VI, according to the namespace templates in Table II and Table III.

There are different access control mechanisms to enforce the namespace policy. Since Tomcat supports RBAC (Role-Based Access Control) model (Sandhu 2000), we adopt this model for access control. The basic idea of RBAC model is to decide which users (cloud users and cloud providers) can access what namespaces based on the roles of the user. More formally, given a set of users \( U \), a set of roles \( R \), and a set of namespaces \( N \), a RBAC model determines the mapping \( U \rightarrow N \) by composing the two independent mappings: 1) \( U \rightarrow R \) maps the users \( (U) \) to the roles \( (R) \); and 2) \( R \rightarrow N \) maps the roles \( (R) \) to the namespaces \( (N) \).

Figure 10 illustrates an architecture where a container factory dynamically configures the RBAC databases used to authorize users. The RBAC databases are set up by the solid arrows as follows:

1. The factory accepts commands from user \( u \) to create task containers for job \( j \) from the task images.
2. The factory authenticates \( u \), creates a role \( r(u,j) \) for \( u \), and saves \( u \rightarrow r(u,j) \) in database \( U \rightarrow R \).
3. The factory asks Docker to launch the containers \( C(j) \) for job \( j \).
4. The factory assigns the namespaces \( N(C(j)) \) to \( r(u,j) \) and save \( r(u,j) \rightarrow N(C(j)) \) in the \( R \rightarrow N \) database (see Table VI for examples of \( N(C(j)) \)).

When user \( u \) adds a new job \( j \), a new role \( r(u,j) \) is created for job \( j \) and its tasks. When a job \( j \) is removed, the corresponding role \( r(u,j) \) is also removed. However, when user \( u \) adds or removes tasks for job \( j \), no change to roles is needed and we only need to change the namespaces \( N(C(j)) \) associated with job \( j \). When a task container migrates to a new server and assumes a new identity, it is treated as removing the old task and adding a new one.

RBAC databases are accessed by the J-RON and T-RON to control client access as shown by the dashed arrows in Figure 10:
5. RON SERVICE COMPOSITION

RON REST APIs normalize the access to individual cloud resources through a set of primitive HTTP operations, but they do not provide a mechanism for a client to concurrently control a large number of RON resources with a few operations. Programming distributed REST resources based on primitive HTTP operations is quite difficult for developers and a high-level service composition language and runtime engine is needed. For this purpose, we choose functional programming with monad, because it has the following advantages over the alternative approaches:

- Concurrency: a functional program can be executed concurrently as it either has no side-effects or can isolate the side-effects with monads (Abelson 1996).
- Efficiency: a functional program can be executed efficiently as it removes the data bottleneck of imperative programming (Backus 1977).
- Dynamism: RON resource identifications and operations unknown at design time can be dynamically computed at runtime by monads and functions according to control criteria, such as task priority, affinity, deadline, network conditions and server capacities.
- Composability: universal compositions of functions and monads can be defined and carried out by a few generic algebraic rules (Wadler 1992).
- Transmission: functional programs tend to be small in size and therefore easy to transmit.

In our composition framework, REST resources are treated as distributed objects with local states (Abelson 1996) that can be accessed by the GET, PUT, POST, and DELETE operations. Since the output of these methods depends on the resource states, they cannot be modeled as pure functions whose output only depends on the input. For this reason, our composition framework uses monads to encapsulate dynamic resource identifications, operations and controls on the REST resources. A collection of monads arranged in a certain way is called a composition program, and it can be evaluated at runtime to a workflow that actually interacts with the REST resources. The overall architecture of our framework is depicted in Figure 11, where the Composition Engine accepts a composition program p through a REST API, evaluates p to a workflow a, and executes a to access the REST resources through the REST clients implemented in certain programming language, e.g. Java.

More formally, the framework can be represented by a 6-tuple \((T, P, W, A, \text{eval}, \text{exec})\):

- \(T\): a set of primitive and composite types defined by a type system.
- \(P\): a set of programs whose types are defined by \(T\).
- \(W\): the REST resources, proxies, gateways, and caches that programs in \(P\) can access.
- \(A\): a set of workflows that access \(W\) through network protocols, such as HTTP 1.1.
- \(\text{eval}: P \rightarrow A\): the procedure that evaluates a program in \(P\) to a workflow in \(A\).
- \(\text{exec}: A \rightarrow T\): the procedure that executes a workflow in \(A\) to produce a result in \(T\), which can be success or faults.

![Figure 11. Architecture of Composition Framework](image-url)

The separation of procedures \(\text{eval}\) and \(\text{exec}\) allows the framework to change the execution environment, for example from Java to Python, without changing the evaluation procedure or the REST API.

A program in \(P\) is composed from three kinds of primitive expressions:

- \(\text{value}\): an expression that evaluates to itself, such that \(\text{eval}(\text{value}) = \text{value}\);
- \(\text{function}\): an expression that evaluates to a mapping between types, such that \(\text{eval}(\text{function}) = \text{t}_1 \rightarrow \text{t}_2\), where \(\text{t}_1, \text{t}_2 \in T\);
- \(\text{monad}\): an expression that evaluates to a workflow in \(A\), such that \(\text{eval}(\text{monad}) = a \in A\).

For example, expression \(v = \text{http://h/a/b}\) represents a value whose type is URI.

Expression \(u(x,y) = \text{http://h/[x]/[y]}\) represents a URI template as a function of type \(\text{string} \rightarrow \text{string} \rightarrow \text{URI}\).

Expression \(u(a,b) = u(a,b)\) represents a program composed from function \(u\) and strings \(a\) and \(b\), and we have \(\text{eval}(u(a,b)) = v\), the URI expression defined above.

The composition of two primitive expressions \(x\) and \(y\) is denoted by a polymorphic binary operator \(x \Delta y\), read as \(x\) meet \(y\). Table VII enumerates compositions permitted by the core language.

To allow order-free compositions, operator \(\Delta\) must be associative such that \((x \Delta y) \Delta z = x \Delta (y \Delta z)\) for all the cases. If \(f\) is a function, \(u\) and \(v\) are values, then the associativity requires that \((f \Delta u) \Delta v = f \Delta (u \Delta v)\). The associativity always holds.
holds because \( f(u,v) \) and \( f([u,v]) \) are semantically equivalent expressions:

1. \( \Delta f(u) \Delta v = f(u) \Delta v = f(u,v) \);
2. \( f \Delta [u,v] = f \Delta [u,v] = f(u,v) \).

**Table VII. Composition Rules for the Primitives**

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x \Delta y )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
<td>([x,y])</td>
<td>list ( x ) and ( y )</td>
</tr>
<tr>
<td>function</td>
<td>value</td>
<td>( y(x) )</td>
<td>function application</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>( x(y) )</td>
<td>function application</td>
</tr>
<tr>
<td>monad</td>
<td>function</td>
<td>( map(y)(x) )</td>
<td>apply function ( y ) to monad ( x )</td>
</tr>
<tr>
<td>function</td>
<td>monad</td>
<td>( map(x)(y) )</td>
<td>apply function ( x ) to monad ( y )</td>
</tr>
<tr>
<td>monad</td>
<td>monad</td>
<td>( join(x,y) )</td>
<td>join monads ( x ) and ( y )</td>
</tr>
<tr>
<td>value</td>
<td>monad</td>
<td>( join([unit(x),y]) )</td>
<td>join the monad from ( x ) with ( y )</td>
</tr>
<tr>
<td>monad</td>
<td>value</td>
<td>( join([x,unit(y)]) )</td>
<td>join ( x ) with the monad from ( y )</td>
</tr>
</tbody>
</table>

To allow order-free compositions, operator \( \Delta \) must be associative such that \( (x \Delta y) \Delta z = x \Delta (y \Delta z) \) for all the cases. If \( x \) is a function, \( u \) and \( v \) are values, then the associativity requires that \( f \Delta (u \Delta v) = f \Delta (u \Delta v) \). The associativity always holds because \( f(u,v) \) and \( f([u,v]) \) are semantically equivalent expressions:

3. \( f \Delta (u \Delta v) = f \Delta (u \Delta v) = f(u,v) \);
4. \( f \Delta (u \Delta v) = f \Delta (u \Delta v) = f(u,v) \).

For other cases, the associativity also holds from the Monad rules. Notice that the meet operator is commutative in some cases but not always, i.e. \( x \Delta y = y \Delta x \) does not hold for value-value, monad-monad, value-monad and monad-value compositions, because a list is not commutative.

The polymorphic meet operator between the primitives gives us the freedom to compose lists of primitives using additional polymorphic algebraic operators. A primitive \( x \) can be composed by dot-product \(*\) with a list of primitives \( y=\{y_1, ..., y_n\} \), denoted by \( x*y \), to produce a new list monad as follows:

\[
x*y=\{x*y_1, ..., x*y_n\}.
\]

This definition is recursive so that it works for nested lists. These recursions will eventually reduce to the meet operator on the primitives in Table VII.

Two list monads \( x=\{x_1, ..., x_n\} \) and \( y=\{y_1, ..., y_n\} \) of length \( n \) can be composed by \( x*y \) to produce a new list monad of length \( n \) as follows:

\[
x*y=\{x_1*y_1, ..., x_n*y_n\}.
\]

Two list monads \( x=\{x_1, ..., x_n\} \) and \( y=\{y_1, ..., y_n\} \) can also be composed by cross-product \( x*y \) to produce a list monad of length \( n \) as follows:

\[
x*y=\{x_1*y_1, ..., x_n*y_n\}.
\]

Since the meet operator is associative, the dot-product and cross-product operators are also associative:

1. \( (x*y)^z = x*(y*z) \);
2. \( (x*y)*z = x*(y*z) \).

It is evident that \( x*y = y*x \) and \( x*y = y*x \) also hold for cases where the meet operators are commutative. The precedence of these operators is: \( <> \Delta >* >\times \).

The four polymorphic operators, \( \Delta \), \( * \), \( \times \) and \( \ast \), allow us to create concise and dynamic REST service compositions based on three types of monads: 1) URI monad: a list of URI templates \( [u(...)]\{functions\} \) evaluated to absolute URIs by functions at runtime; 2) operation monad: a list of operation templates \( [o(...)]\{functions\} \) evaluated to HTTP operations by functions at runtime; and 3) control monad: a tree of parallel \( par[...] \) and sequential \( seq[...] \) control templates evaluated to workflows by functions at runtime. Two examples are used to illustrate the composition programs and evaluation process, while the formal definitions of these monads and more examples can be found in (Li, L. 2015).

**Example 1:** Suppose we want to decrease the CPU and memory usage of some idle tasks and give them back to the tasks whose workloads are high, we can write a program as follows:

\[
p1=seq[
\{[j1,jid],[t1,tid] \} \rightarrow \\{par[o(put(u,q),q \leftarrow g1(\ldots))],
\{[j2,jid],[t2,tid] \} \rightarrow \\{par[o(put(u,q),q \leftarrow g2(\ldots))]
\}]
\]

Here functions \( f1 \) and \( f2 \) find the tasks, and functions \( g1 \) and \( g2 \) determine the proper capacity limits for the tasks, as shown in the previous examples. The program evaluates to:

if

\[
f1=[\{cpu=10,mem=10\},\{cpu=20,mem=20\}],
\{cpu=30,mem=30\},\{cpu=40,mem=40\} \]

then

\[
eval(p1)=seq[
\{par
\{\ldots t1,put,[cpu=10,mem=10]\},\{\ldots t2,put,[cpu=20,mem=20]\}\}
\{par
\{\ldots t3,put,[cpu=30,mem=30]\},\{\ldots t4,put,[cpu=40,mem=40]\}\}
\}]
\]

This program will execute the first groups of operations in parallel to decrease the capacities of the idle tasks, and after they are successful, execute the second group of operations in parallel to increase the capacities of the busy tasks.

**Example 2:** If we want to move two sets of busy tasks in parallel to new machines and reset their CPU and memory capacities in the sequential order, we can use a program like this:

\[
p2=par[
\{h-move*par[u(\{f1\})*xseq[o(u1,put,g1),o(u2,put,g2)]]
\{h-move*par[u\{f2\})xseq[o(u1,put,g3),o(u2,put,g4)]]
\}
\]

This program consists of the following functions:
move: operation $o(u, move, target) \to hypertext$ moves a task to the target machine;

$h$: function that finds control hyperlinks from a hypertext;

$f1, f2$: functions that find the busy tasks;

$g1, g2, g3, g4$: functions that determine the task capacities on the target machine;

$u1$: function $u1: u \to \{u/\text{cpu}\}$ expands URI $u$ with path for CPU; and

$u2$: function $u2: u \to \{u/\text{mem}\}$ expands URI $u$ with path for memory.

Function $h$ extracts the control URI from the hypertext, and if it is not there, follows the hyperlinks to discover the control URI. Suppose operation $o(t1, move, z2)$ returns this hypertext:

```
200 OK HTTP 1.1
location: http://z2/jobs/j1/tasks/t1
control: http://z2/jobs/j1/tasks/t1/control
```

Applying $h$ to the hypertext, we have:

```
h \circ (o(t1, move, z2)) = http://z2/jobs/j1/tasks/t1/control.
```

The above program evaluates the following control monad, after the successful move operations:

```
if
f1 = [[j1,t1],[j2,t2]],
f2 = [[j3,t3],[j4,t4]],
g1 = [cpu=10], g2 = [mem=20],
g3 = [cpu=30],  g4 = [mem=40],

then

eval(p2) = par
seq
  o(...t1/control/cpu, put,[cpu=10]),
o(...t1/control/mem, put,[mem=20])
seq
  o(...t2/control/cpu, put,[cpu=10]),
o(...t2/control/mem, put,[mem=20])
seq
  o(...t3/control/cpu, put,[cpu=30]),
o(...t3/control/mem, put,[mem=40])
seq
  o(...t4/control/cpu, put,[cpu=30]),
o(...t4/control/mem, put,[mem=40])

).
```

The program contains 4 sequential groups that execute in parallel.

The Composition Engine exposes a REST API to accept composition programs encoded in XML. A client can submit (POST) a XML program to the REST API, which will call the `eval` procedure and, if successful, return a distinct hyperlink to control the workflow execution. The client can run the workflow many times by sending POST messages to the control resource, which will call the exec procedure and return execution status in the response.

6. Prototype System and Experiments

The described RON REST APIs and the Monad service composition engine have been implemented in Java with the help of the RC-Java tool described in Section III. The experimental environment for both prototypes involved a Linux server machine and a Windows 7 client machine connected by LAN as shown in Figure 12. The browser at the client machine was used to issue individual REST API requests to test the effect of the REST API to control and manage RON running on the server machine, while the JMeter is used to run the performance tests.

5.1 RON Prototype and Experiments

The RON REST APIs were deployed on Tomcat servers, and the prototype system was integrated with Docker running on Linux servers.

Figure 13 shows the effect of increasing the CPU cores from 1 to 3 for a container using the REST API, as observed by cAdvisor, an open-source container monitoring tool (cAdvisor 2015). The blue curve indicates that the container uses core 0 at the beginning. After we used the REST API to set the cores to 3, the container immediately begins to use three cores 0, 1 and 2, while the load on core 0 was decreased.

Figure 14 illustrates the effect of memory limit change for a container observed by cAdvisor. At beginning the memory limit of the container was kept at the maximum, and after we used the REST API to set it to 1.9GB, the total memory (blue curve) immediately dropped down to the new limit.

To test the performance of the REST API, the RON REST APIs and the Tomcat that hosted them were packaged into a Docker image and ran as a Docker container with all the default configurations. We used JMeter to simulate 5 and 10 concurrent REST clients, while each REST client sent 20 different GET and PUT requests to the REST APIs. The average client response time (millisecond) and the total RON based REST API system processing throughput (#request/s) were calculated. In each session, we also recorded the REST API processing time (millisecond) for each request at the server side. To test the task workload impact on the REST API performance, we ran 7, 22, and 32 Docker containers to simulate the task loads on the same Linux server, whereas the RON based REST API server was running inside the separate dedicated container.
Figure 15 shows the performance of 5 concurrent clients (time on the left y-axis and throughput on the right y-axis) when the number of active task containers increased (on x-axis). The results show that the average client response time to the RON container increased linearly from 56ms to 138ms and 178ms for 7, 22 and 32 task containers respectively. The gap between the average client response time and server time is 5ms, which accounts for the average network latency and Tomcat processing time.

Three REST service composition programs in cloud computing were used to test the performance of the workflow engine. The REST API runs inside the Tomcat on a Linux server and it accepts REST composition programs that adjust the CPU, memory, and I/O resources used by Docker containers (Docker 2015), where effects of resource management by these operations on the running containers were observed and monitored by cAdvisor (cAdvisor 2015).

In this experiment, as the number of concurrent client increased from 5 to 10, the overall RON based REST API server system throughput increased by 31% on the average, while the average client response time increased by 58% and the average server processing time increased by 53%.

These experiments showed that the average client and server response time increased linearly in most cases with the number of concurrent clients, and it increased the task workloads under the default configurations for Tomcat and Linux. We believe that the performance can be further improved with tuning and optimization of the Linux and Tomcat configurations as well as REST API implementations.

5.2 COMPOSITION PROTOTYPE AND EXPERIMENTS

We used JMeter to simulate the concurrent HTTP clients that submit and execute composition programs. Since the composition REST API is typically accessed by a few clients who have proper management privileges, we launched 1, 5, and 10 concurrent clients at each test session to simulate this situation. For each session, the average response time (client) and the system throughput were reported by JMeter, while the server processing time (server) was reported by Java methods. All time units in the experiments were measured in milliseconds. Each client submitted and ran three composition programs:

- **update_cpuset** (1.28KB): change the CPU cores used by 5 containers and their parent control groups in sequence;
- **update_multiple** (1.23KB): change the CPU usage shares and the memory limits used by 5 containers in parallel;
- **update_usehierarchy** (1.82KB): stop all containers in parallel, set the memory use_hiearchy
parameters for them, and then restart the containers in parallel.

The first experiment tested the time of the eval procedure, which converts XML programs to Java workflow objects. Each client in a test session repeated 6 requests 10 times to submit and delete the above 3 programs. As the result, the total requests for 1, 5, and 10 concurrent clients were 60, 300, and 600 respectively. The test results are depicted in Figure 17, Figure 18, and Figure 19.

In these tests, the client response time and server processing time decreased and the throughputs increased, as the number of concurrent clients increased. This could be due to the memory caches that save frequently used Java objects in memory as more requests repeat themselves in the experiments. Since the server has 12 physical CPU cores, increasing the number of concurrent clients did not slow down the server processing time as they were not overloaded.

The second set of experiments tested the time of the exec procedure, which involves computation inside the Java engine as well as external calls to the composed REST resources in the same machine. Each client in a test session repeated 3 requests to run the 3 workflows for 10 times. As the result, the total requests for 1, 5, and 10 clients were 30, 150, and 300 respectively. The test results are shown in Figure 20, Figure 21, and Figure 22.

In these tests, the client response time and server processing time increased linearly with the number of concurrent clients, except for workflow update_multiple, which only contains parallel operations that are implemented by Java Threads. With 1 client, the average client and server time for update_multiple was 15ms, and
7ms respectively. For update_cpuset, it was 287ms and 279ms, and for update_usehierarchy, it was 2401ms and 2391ms. The difference is partly due to the structural differences between the workflows and partly due to our current implementation strategy that uses Java threads for parallel operations. This also explains the execution time differences between the workflows as the number of concurrent client increases.

The overall test results indicate that the proposed approach is feasible and there is room to fine-tune and improve the implementations.

7. CONCLUSIONS
The main contributions of this paper are summarized below:

- A REST service framework based on Resource-Oriented Network (RON) for RaaS Cloud that decouples the representation plane, control plane, and data plane for efficient resource management.
- The RON based REST APIs to normalize the access to heterogeneous fine-grained resources, including CPU, memory and storage, at the levels of tasks, jobs, containers, servers, and clusters; and it has been implemented and tested with Docker containers.
- A dynamic RBAC mechanism to control the access to REST API namespaces where the roles and permissions are created when the containers are launched to enhance the security.
- A lightweight REST service composition framework to dynamically construct REST workflows using monads at runtime to concurrently control a large number of REST resources with a few operations.

The monadic composition language currently does not support more advanced features, such as conditions and exceptions, as our use cases do not require them. We anticipate such features will become necessary as we apply the framework to more complicated situations.

For future work, we plan to expand the functionalities of RON and integrate it with automated agents for resource allocation, scheduling, scaling, and trading into elastic machines in server-based and disaggregated RaaS clouds.

8. ACKNOWLEDGMENT
The authors would like to thank Tony Tang for implementing and testing the prototypes described in this paper.

9. REFERENCES

Dr. Li Li received his Ph.D. in computer sciences from University of Alabama at Birmingham, USA in 1995, and M.S. in computational linguistics from Huazhong University of Sciences and Technology, China in 1987. He joined Huawei Shannon IT Lab in 2012 and his current research interest includes web services, cloud computing and software-defined networking. He has published over 70 conference and journal papers and 1 book on Artificial Intelligence. He currently holds 12 US patents. Dr. Li is a member of IEEE and ACM, and he was the editor of 2 ISO/ECMA CSTA III standards and made significant contributions to W3C WS-RA standard suite.

Dr. Wu Chou currently is VP and Chief Technology Officer (CTO) of Network and Enterprise Communications product line of Huawei. He graduated from Stanford University with four advanced degrees in science and engineering. He joined AT&T Bell Labs after obtaining his Ph.D. degree in electrical engineering, and he continued his professional career from AT&T Bell Labs to Lucent Bell Labs and Avaya Labs before joining Huawei. He published over 200 journal and conference papers, and holds over 40 US and international patents with many additional patent applications pending. Dr. Chou is an IEEE Fellow. He served as editor and area expert for multiple international standards at W3C, ECMA, ISO, ETSI, etc. He was an editor of IEEE Transactions on Services Computing (TSC), IEEE TSC Special Issue on Cloud Computing, IEEE Transactions on Audio and Language Processing, and Journal of Web Services Research.

Dr. Min Luo is currently the Head and Chief Architect of the Advanced Networking at Huawei’s Shannon (IT) Lab, leading the research and development in Software Defined Networking (SDN) and other future networking initiatives. He served as Chief Architect for IBM’s Global Business Solution Center –GCG, Industry Solutions, and Center of Excellence for Enterprise Architecture and SOA for more than 11 years. He also worked as Senior Operations Research Analyst, Senior Manager and Director of Transportation Network Planning and Technologies for two Fortune 500 companies for 7 Years. He is a senior member of IEEE, he has been serving on the organizing committee for IEEE’s ICWS and SCS/CC Conferences, chaired sessions, presented several tutorials on SOA and Enterprise Architecture and their best practices and gave lectures at the Service University. He has served as adjunct professors in several US and Chinese universities since 1996. He obtained his Ph.D. in Electrical Engineering from Georgia Institute of Technology in 1992. He also held a M.S. (1987) and B.S (1982) in Computer Science.

Authors

Dr. Li Li

Dr. Min Luo


